**Утилитарные типы (utility types)** в TypeScript — это встроенные типы, которые позволяют изменять или комбинировать другие типы. Они помогают работать с объектами, функциями и другими структурами данных, делая типизацию гибкой и мощной. Эти утилиты упрощают написание сложных типов, поскольку позволяют извлекать, модифицировать или создавать новые типы на основе уже существующих.

Давайте подробно рассмотрим основные утилитарные типы с примерами:

**1. Partial**

Создает тип, в котором все свойства типа T становятся опциональными.

type User = {

  id: number;

  name: string;

  age: number;

};

// Все свойства теперь опциональны

type PartialUser = Partial;

const user: PartialUser = {

  name: "John" // Необязательно указывать остальные свойства

};

**2. Required**

Делает все свойства типа T обязательными. Противоположность Partial.

type User = {

  id?: number;

  name?: string;

};

// Теперь все свойства обязательны

type RequiredUser = Required;

const user: RequiredUser = {

  id: 1,

  name: "John"

};

**3. Readonly**

Создает тип, в котором все свойства становятся только для чтения (их нельзя изменить после инициализации).

type User = {

  id: number;

  name: string;

};

// Все свойства только для чтения

type ReadonlyUser = Readonly;

const user: ReadonlyUser = {

  id: 1,

  name: "John"

};

// user.id = 2; // Ошибка, потому что свойство только для чтения

**4. Record**

Создает тип объекта, где ключи определяются типом K, а значения — типом T.

type Roles = "admin" | "user" | "guest";

// Ассоциируем роли с числовым значением

type RoleCounts = Record;

const roles: RoleCounts = {

  admin: 1,

  user: 10,

  guest: 100

};

**5. Pick**

Создает новый тип, выбирая только указанные свойства K из типа T.

type User = {

  id: number;

  name: string;

  email: string;

};

// Оставляем только id и name

type UserInfo = Pick;

const user: UserInfo = {

  id: 1,

  name: "John"

  // email: "[john@example.com](mailto:john@example.com)" // Ошибка, так как мы не включили email

};

**6. Omit**

Создает новый тип, исключая указанные свойства K из типа T. Это противоположность Pick.

type User = {

  id: number;

  name: string;

  email: string;

};

// Исключаем email

type PublicUser = Omit;

const user: PublicUser = {

  id: 1,

  name: "John"

};

**7. Extract**

Создает тип, состоящий только из тех частей T, которые совместимы с U.

type Data = string | number | boolean;

// Оставляем только string и number

type StringOrNumber = Extract;

const value: StringOrNumber = "Hello"; // Либо число, либо строка

// const value: StringOrNumber = true; // Ошибка, так как boolean исключен

**8. Exclude**

Выбирает из типа T все, что не совместимо с U. Противоположность Extract.

type Data = string | number | boolean;

// Исключаем string и number

type OnlyBoolean = Exclude;

const value: OnlyBoolean = true;

// const value: OnlyBoolean = "Hello"; // Ошибка

**9. NonNullable**

Удаляет из типа T все null и undefined.

type Data = string | null | undefined;

// Убираем null и undefined

type NonNullableData = NonNullable;

const value: NonNullableData = "Hello";

// const value: NonNullableData = null; // Ошибка

**10. Parameters**

Возвращает тип списка аргументов функции T.

type GetUser = (id: number, name: string) => void;

// Тип аргументов функции

type Args = Parameters;

const args: Args = [1, "John"];

**11. ReturnType**

Возвращает тип результата функции T.

type GetUser = () => { id: number; name: string };

// Тип возвращаемого значения

type Result = ReturnType;

const user: Result = {

  id: 1,

  name: "John"

};

**12. ConstructorParameters**

Возвращает тип аргументов конструктора класса.

class User {

  constructor(public id: number, public name: string) {}

}

// Тип аргументов конструктора

type ConstructorArgs = ConstructorParameters;

const args: ConstructorArgs = [1, "John"];

**13. InstanceType**

Возвращает тип экземпляра класса.

class User {

  id: number;

  name: string;

}

// Тип экземпляра класса

type UserInstance = InstanceType;

const user: UserInstance = new User();

Полезная комбинация утилитарных типов

Вы можете комбинировать несколько утилитарных типов для создания более сложных типов:

type User = {

  id: number;

  name: string;

  email: string;

};

type SafeUser = Readonly>>;

const user: SafeUser = {

  id: 1

};

// user.id = 2; // Ошибка, так как свойства только для чтения

**Заключение**

Утилитарные типы в TypeScript мощно расширяют возможности типизации. Они позволяют создавать производные типы, которые упрощают разработку и сопровождение кода. Используйте их, чтобы избежать избыточности и сделать код более выразительным